Санкт-Петербургский Государственный Университет

Математико-механический факультет

Кафедра системного программирования

Пыхтин Александр Васильевич

Разработка механизма внесения контролируемых изменений в порядок многопоточного исполнения библиотеки LinCheck

Курсовая работа

Научный руководитель:
старший преподаватель Кириленко Я.А.

Санкт-Петербург

2016

Оглавление

Введение

Параллельность в современных системах

В настоящее время производители процессоров увеличивают их вычислительные мощности за счет увеличения числа ядер. Разбиение программы на независимые части и их выполнение одновременно на разных ядрах позволяет уменьшить общее время её выполнения. Вычисления, выполняемые таким образом, называют параллельными, и программу, выполняющую параллельные вычисления, в свою очередь, называют параллельной.

В последовательной модели программирования инструкции компьютерной программы выполняются поочередно. Все задачи выполняются по порядку, и каждая из них должна ожидать своей очереди. Каждая задача, прежде чем приступить к своей работе, должна ожидать до тех пор, пока не получит результатов выполнения предыдущей.

В отличие от последовательного исполнения, в параллельной программе в каждый момент времени могут одновременно выполняться несколько инструкций. Программа разбивается на множество параллельных задач. Несколько задач могут одновременно начать выполнение на любом ядре процессора без какой бы то ни было гарантии того, что некоторая задача завершится первой, или все они завершатся в определенном порядке.

Исходя из вышесказанного, при разработке параллельных программ могут возникать определенные трудности. Они заключаются в том, что разработчик не может знать точно, в какой последовательности эти инструкции будут выполняться. Кроме того, при каждом запуске последовательность их исполнения может отличатся. К тому же, сами потоки работают абсолютно независимо друг от друга, что перекладывает всю ответственность за синхронизацию выполнения и обеспечение связи между потоками программиста. При некорректно реализованной связи или синхронизации

обычно возникает четыре типа проблем. Более того, из - за неопределенной последовательности выполнения инструкций, обнаружение этих ошибок усложняется.

Типы возникающих проблем

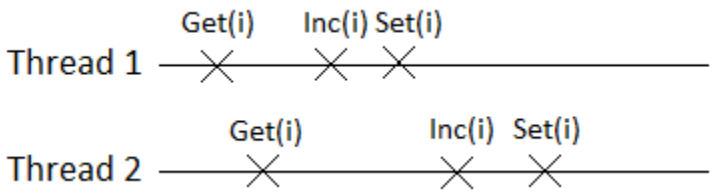
DataRace

Если несколько задач пытаются одновременно изменить некоторую общую область данных, то может возникнуть ситуация, называемая DataRace. Возникновение подобной ситуации может привести к появлению в памяти объектов, находящихся в неконсистентном состоянии.

Например, оба потока пытаются сделать инкремент переменной. Инкремент выполняется в 3 этапа:

1. Взятие значения переменной;
2. Увеличение значения на единицу;
3. Запись значения в переменную.

Каждый из этих этапов является атомарным. Если подобный инкремент будет вызван из нескольких потоков, то есть вероятность, что несколько потоков одновременно получат одинаковое значение, которое увеличат на 1 и запишут в переменную. Такие действия приведут к некорректному результату в инкриментируемой переменной. Пример подобного исполнения показан на рисунке.

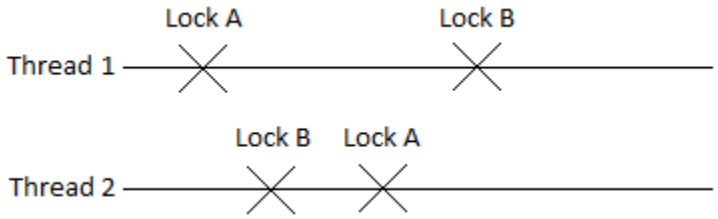


Эта проблема разрешается с помощью доступа к объекту только одного потока одновременно, но решение проблемы приводит к другой, часто возникающей ошибке.

Deadlock

Если несколько задач пытаются захватить доступ одновременно к нескольким объектам, то может возникнуть ситуация, называемая DeadLock.

DeadLock – ситуация в многозадачной среде при которой несколько процессов находятся в состоянии бесконечного ожидания ресурсов, занятых самими этими процессами. Пример Deadlocka показан на рисунке:



Поток номер 1 захватывает ресурс A, после чего хочет захватить ресурс B.
Поток номер 2 захватывает ресурс B, после чего хочет захватить ресурс A. Ситуация, при которой поток номер 1 будет ожидать освобождения ресурса B, а поток номер 2 освобождения ресурса A называется взаимной блокировкой.

LiveLock

LiveLock – Ситуация, схожая с DeadLock. Однако в отличие от DeadLock потоки не застревают на месте, а продолжают совершать какие-то действия, которые не приводят к конечному результату(зацикливается).

Трудности организации связи

Иногда один поток может не увидеть изменений, сделанных в другом потоке или увидеть их не в том порядке. Подобная ситуация может возникнуть при недостаточных средствах связи между потоками, что приводит к появлению вычислений, основанных на неактуальных данных.

Обзор инструмента

LinCheck

Для проверки корректности многопоточных структур данных был создан инструмент LinCheck. Стоит отметить, что аналогом понятия «корректность» для многопоточных структур данных является понятие «линеаризуемость». Линеаризуемость — свойство программы при котором результат некоторого параллельного выполнения операций соответствует некоторому последовательному выполнению. LinCheck – инструмент, проверяющий структуру данных на линеаризуемость. Инструмент ищет набор операций над структурой, параллельное исполнение которых не может быть объяснено последовательным. Что позволяет найти ошибки типа DataRace и Rase Condition.

Описание метода проверки

Выбирается небольшое количество потоков, в которых будет тестироваться структура данных. Далее создается тестовых набор операций. Он представляет собой фиксированный набор операций, которые будут вызваны над объектом. Для этого набора генерируются все возможные последовательные исполнения для объекта и фиксируются результаты каждого из них. Следующим шагом является генерация многопоточных исполнений набора операций. Для каждого многопоточного исполнения фиксируется результат выполнения. Далее выполняется его поиск среди результатов последовательного исполнения. Если совпадение найдено, то это значит, что многопоточное исполнение корректно и можно переходить к следующему. Если же совпадение не найдено, то результат параллельного исполнения, невозможно получить при последовательном исполнении. Такая ситуация означает, что была обнаружена ошибка. (многопоточное исполнение нелинеаризуемо). В дальнейшем эту информацию можно будет использовать для поиска ошибки в реализации структуры данных.

Особенности реализации

Для передачи данных анализатору используется механизм аннотаций. Чтобы протестировать структуру данных, требуется создать класс и специальным образом его разметить с помощью аннотаций. По этому классу будет генерироваться набор тестовых сценариев. Для этого с помощью библиотеки ASM генерируется java-байткод, который реализует класс с вызовом всех необходимых операций. После того, как тестовый набор был создан, необходимо сгенерировать для него многопоточные исполнения. Для этого созданный набор исполняется большое количество раз в многопоточном режиме. Результаты исполнения сохраняются в специальный массив. Важным условием, влияющим на работоспособность анализатора, является качество генератора многопоточных исполнений. Если при запусках будет преобладать какой-то конкретный набор исполнения команд, то значимых результатов не получится. Нужно стремиться к тому, чтобы операции над структурой данных исполнялись максимально разнообразно, покрывая все возможные состояния структуры, при этом операции должны исполняться действительно параллельно. В настоящий момент используется случайная задержка перед выполнением каждой операции, что приводит к сильной неравномерности получения различных результатов.

Постановка задачи

Требуется разработать механизм внесения контролируемых изменений в порядок многопоточного исполнения для получения равномерного распределения результатов выполнения тестовых сценариев. При этом важное значение имеет параллельное исполнение операций над структурой данных, а также сохранение изначально заложенной в структуру логики работы с многопоточностью. Разработанный механизм позволит обеспечить большее покрытие состояний тестируемой структуры данных за то же время, а также повысит вероятность обнаружения ошибки в линеаризуемости операций.

Обзор вариантов решения

Случайные задержки

За переключение контекста и время, отданное каждому потоку, отвечает системный планировщик. Манипулировать выполнением на его уровне не представляется возможным, что приводит к некоторым трудностям. Мы не знаем, когда будет вызван тот или иной метод, какой поток начнет работу первым и когда произойдет переключение выполнения. Именно эту проблему отчасти решает создание случайных задержек перед вызовом тестируемого метода. К сожалению, подобный подход не позволяет управлять выполнением методов, что приводит к случайному распределению полученных результатов. Добавление больших задержек позволит манипулировать порядком исполнения, но приведет к вырождению параллельного выполнения в линейное. Очевидно, что в таком случае, ошибок, связанных с параллельным выполнением обнаружить не удастся.

Можно попытаться высчитать время, необходимое для выполнения каждого из тестируемых методов, или время, которое проходит до вызова каждого конкретного метода в потоке, но подобные попытки не позволят манипулировать последовательностью исполнения, поскольку его порядок и время, отведенное на работу каждого потока, меняется при каждом новом запуске и независимо от предыдущих.

Добавление синхронизатора

Следующей идеей является добавление синхронизатора в генерируемый код. Перед вызовом каждого тестируемого метода вставляется синхронизатор, который в случае необходимости будет ожидать, пока остальные потоки дойдут до необходимой точки, и только после этого продолжит выполнение. Однако попытка синхронизировать потоки средствами Java приводит к выполнению отношения happens before. «happens before» - отношение строгого частичного порядка, введённое между атомарными командами, которое означает, что вторая

команда будет «в курсе» изменений, проведённых первой. Выполнение подобного отношения при тестировании приведет к невозможности обнаружения ошибок недостаточной связи между потоками и к уменьшению обнаружения ошибочных случаев. Чтобы отношение happens before не выполнялось, нужно неким образом обмануть компилятор при синхронизации. Это возможно сделать, написав синхронизатор на другом языке программирования и использовав его через Java Native Interface. Таким образом, компилятору ничего не известно о синхронизации потоков и отношение happens before при синхронизации не выполняется.

Реализация

Для манипуляции над порядком исполнения команд при генерации тестового набора результатов, запоминается последовательность исполнения команд, приводящая к получению конкретного результата. После чего во время параллельного выполнения эти данные используются в сгенерированных классах для синхронизации выполнения методов. В процессе работы было опробовано два подхода к синхронизации.

Попытка честного параллельного исполнения в спорных местах

С использованием примитива синхронизации типа «Барьер» была предпринята попытка исполнения части методов последовательно, а методы в моментах чередования потоков были выполнены параллельно с использованием случайной задержки перед их вызовом. Это позволило несколько нормализовать гистограмму, но, в связи с неизвестным временем начала выполнения операции и высокой скоростью их исполнения, подобные действия привели к запуску методов в линейном режиме (параллельность выродилась в линейность). Подобный метод не позволяет найти ошибки, связанные с DataRace.

Комбинация последовательного и параллельного кода

Лучший результат дала комбинация последовательного и параллельного исполнения методов со случайными задержками перед их вызовом. В данной реализации после нескольких прогонов параллельного исполнения программа находит результат с наименьшим количеством попаданий, после чего берет последовательность команд, приводящих к этому результату, и выполняет часть из них последовательно, что приводит к большей вероятности попадания в необходимый результат. Остальные команды выполняются в параллельном режиме со случайной задержкой. Если после этого количество попаданий в требуемый результат осталось минимально, то количество команд исполняемых в последовательном режиме увеличивается. После того, как количество попаданий в требуемый результат перестаёт быть минимальным, берется новый

минимальный результат и все начинается сначала. Несмотря на то, что подобная методика может привести к вырождению выполнения к линейному случаю, на практике достаточно выполнить в линейном порядке не более половины команд для высокой вероятности попадания в необходимый результат. Таким образом, подобная методика позволила внести контролируемые изменения в порядок многопоточного исполнения, получить равномерное распределение результатов выполнения тестовых сценариев. параллельность исполнения операций над структурой данных, а так же изначально заложенная в структуру логика работы с многопоточностью была сохранена. Разработанный механизм позволяет обеспечить большее покрытие состояний тестируемой структуры данных за то же время и повышает вероятность обнаружения ошибки в линеаризуемости операций.

Особенности реализации

Для синхронизации выполнения методов с помощью библиотеки ASM, была реализована генерация классов, отвечающих за запуск тестовых методов.

```
public class Generated10 extends Generated {
    public Object testClass;
    private Barier barier;
    public Generated10(Object testClass, Barier barier) {
        this.testClass = testClass;
        this.barier = barier;
    }
    public void process(Result[] res, MethodParameter[][] args, int[] waits, int[] offset) {
        for (int i = 0; i < offset[0]; i++) {
            barier.waitOtherTread();
        }
        try{
            MyRandom.busyWait(waits[0]);
            res[0].setValue(testClass.CallMethod(args[0][0]));
        }catch (Exception e) {
            res[0].setException(e);
        }
...
      barier.waitOtherTread();
    }
}
```

Barier

Barier – класс отвечающий за взаимодействие с синхронизатором, написанном на C++

```
public class Barier {
    static{
        System.load("barier.dll");
    }
    public Barier(int threads){
        init(threads);
    }
    private native void init(int threads);
    public native void waitOtherTread();
}
```

В конструткоре Barier создает синхронизатор, передавая ему информацию о количестве потоков, которые необходимо синхронизировать. При вызове метода waitOtherThread() текущий поток начинает ожидать, пока остальные синхронизируемые потоки не вызовут этот метод. После того, как количество потоков, вызвавших waitOtherTread() совпало с указанным при создании, все потоки одновременно продолжают свое выполнение. Взаимодействие сгенерированных классов с методами, написанными на другом языке программирования, позволило манипулировать порядком запуска операций, не выполняя при этом отношение happens before между потоками.

Изменение API

Для упрощения взаимодействия между библиотекой и пользователем было изменено API, что позволило более простым способом организовать

```
@CTest(iter = 300, actorsPerThread = {"1:3", "1:3"})
@CTest(iter = 300, actorsPerThread = {"1:3", "1:3", "1:3"})
public class QueueTest {
    public Queue<Integer> q;
    @Reset
    public void reload() {
        q = new GenericMPMCQueue(16);
    }


    @Actor(args = {"1:10"})
    public void offer(Result res, Object[] args) throws Exception {
        Integer value = (Integer) args[0];
        res.setValue(q.offer(value));
    }

    @Actor(args = {})
    public void poll(Result res, Object[] args) throws Exception {
        res.setValue(q.poll());
    }

    @Immutable
    @Actor(args = {})
    public void peek(Result res, Object[] args) throws Exception {
        res.setValue(q.peek());
    }
}
```

взаимодействие с тестовой структурой.

Старая версия API

```java
@CTest(iter = 300, actorsPerThread = {"1:3", "1:3"})
@CTest(iter = 300, actorsPerThread = {"1:3", "1:3", "1:3"})
@Param(name = "key", clazz = FloatGenerator.class, opt = {"0", "10", "0.1"})
public class QueueCorrect1 {
    public GenericMPMCQueue<Integer> q;

    @Reset
    public void reload() {
        q = new GenericMPMCQueue(2);
    }

    @Operation
    public boolean offer(@Param(clazz = IntegerGenerator.class) int value) throws Exception {
        return q.offer(value);
    }

    @Operation
    public int poll() throws Exception {
        return q.poll();
    }

    @Test
    public void test() throws Exception {
        assertTrue(Checker.check(new QueueCorrect1()));
    }
}
```

Новая версия API

Изменения претерпели объявления тестируемых методов. В новой версии параметры методов передаются и возвращаются в явном виде. Так же допускается объявление параметров в аннотации класса и использование в дальнейшем лишь имени параметра. К тому же типы принимаемого и возвращаемого значения были расширены с Integer до любых типов. Для примитивных типов были написаны генераторы.

Изменения в генерируемых классах

В исходной версии библиотеки максимальное количество методов в одном потоке равнялось пяти. Данного ограничения хватало для тестирования примитивных структур данных, но с изменением API появилась потребность в увеличении количества методов в потоке. Для этого были внесены изменения в механизм генерации тестовых классов, что позволило повысить предельное количество методов на поток, не увеличивая при этом время выполнения тестовых методов. В настоящее время количество методов в потоке ограничено только размером стека.

Результаты

В результате работы был разработан механизм внесения контролируемых изменений в порядок многопоточного исполнения. Получено равномерное распределение результатов выполнения тестовых сценариев.
Так же были расширены возможности библиотеки для тестирования различных структур данных.

Ниже приведены результаты тестирования корректно реализованной очереди. Каждый из графиков показывает результат выполнения 50_000 раз одного из тестовых наборов данных.
(-10 означает, что возвращаемый тип данных операции - void)
Конфигурация:

№ Потока	Операции в потоке		
1	Put(0)	Get()	
2	Put(9)	Put(9)	Get()
3	Put(5)	Put(7)	

Возможные результаты:

№ Результата	Результат выполнения
1	-10, 0, -10, -10, 9, -10, -10
2	-10, 0, -10, -10, 5, -10, -10
3	-10, 9, -10, -10, 0, -10, -10
4	-10, 5, -10, -10, 0, -10, -10
5	-10, 9, -10, -10, 9, -10, -10
6	-10, 9, -10, -10, 5, -10, -10
7	-10, 5, -10, -10, 9, -10, -10
8	-10, 5, -10, -10, 7, -10, -10
9	-10, 7, -10, -10, 5, -10, -10

№ Результата	Случайные задержки	Задержки с синхронизатором
1	4611	8769
2	2684	5019
3	4541	4476
4	923	4487
5	843	6568
6	10066	6159
7	2675	4461
8	7758	5600
9	15899	4461

№ Потока	Операции в потоке		
1	Put(7)	Put(2)	
2	Put(2)	Get()	Put(6)
3	Put(0)		

№ Результата	Результат выполнения
1	-10, -10, -10, 7, -10, -10
2	-10, -10, -10, 2, -10, -10
3	-10, -10, -10, 0, -10, -10

№ Результата	Случайные задержки	Задержки с синхронизатором
1	3581	16665
2	10790	16682
3	35629	16653

№ Потока	Операции в потоке	
1	Put(5)	Put(2)
2	Get()	
3	Get()	

№ Результата	Результат выполнения
1	-10, -10, 5, 2
2	-10, -10, 2, 5
3	-10, -10, 5, QueueEmptyException
4	-10, -10, QueueEmptyException, 5
5	-10,-10, QueueEmptyException, QueueEmptyException

№ Результата	Случайные задержки	Задержки с синхронизатором
1	942	7728
2	1821	3194
3	11667	3181
4	4733	5831
5	30837	4211

Их представленных графиков мы видим, что результаты выполнения тестовых сценариев стали распределяться более равномерно. Благодаря подобному распределению результатов, библиотека может с большей вероятностью находить ошибки в параллельных структурах данных.

Заключение

Задача тестирования корректности параллельных структур данных не является тривиальной. Хотя реализованный подход внес улучшения в распределение результатов выполнения тестовых сценариев и позволил с большей вероятностью обнаруживать ошибки в параллельных структурах, он не является единственно верным подходом к решению задачи тестирования. В дальнейшем планируется рассмотреть альтернативные методы:

- Автоматическая перемешка тестируемых методов
- Написание различных планировщиков выполнения

Так же планируется попытаться найти решение проблемы взаимодействия с системным планировщиком.

Список литературы

[1] Liang Sheng. The Java Native Interface. — 1999. — URL: https://web.archive.org/web/20120905183616/http://java.sun.com/docs/books/jni/download/jni.pdf.

[2] Shacham Ohad, Bronson Nathan, Aiken Alex. Testing Atomicity of Composed Concurrent Operations. — URL: http://theory.stanford.edu/~aiken/publications/papers/oopsla11b.pdf.

[3] А.А. Евдокимов. Автоматическое тестирование линеари- зуемости реализаций многопоточных структур данных. — 2015. — URL: http://tmpaconf.org/images/pdf/2015/ evdokimov-tsytelov-elizarov-trifanov-automat-testing.pdf.